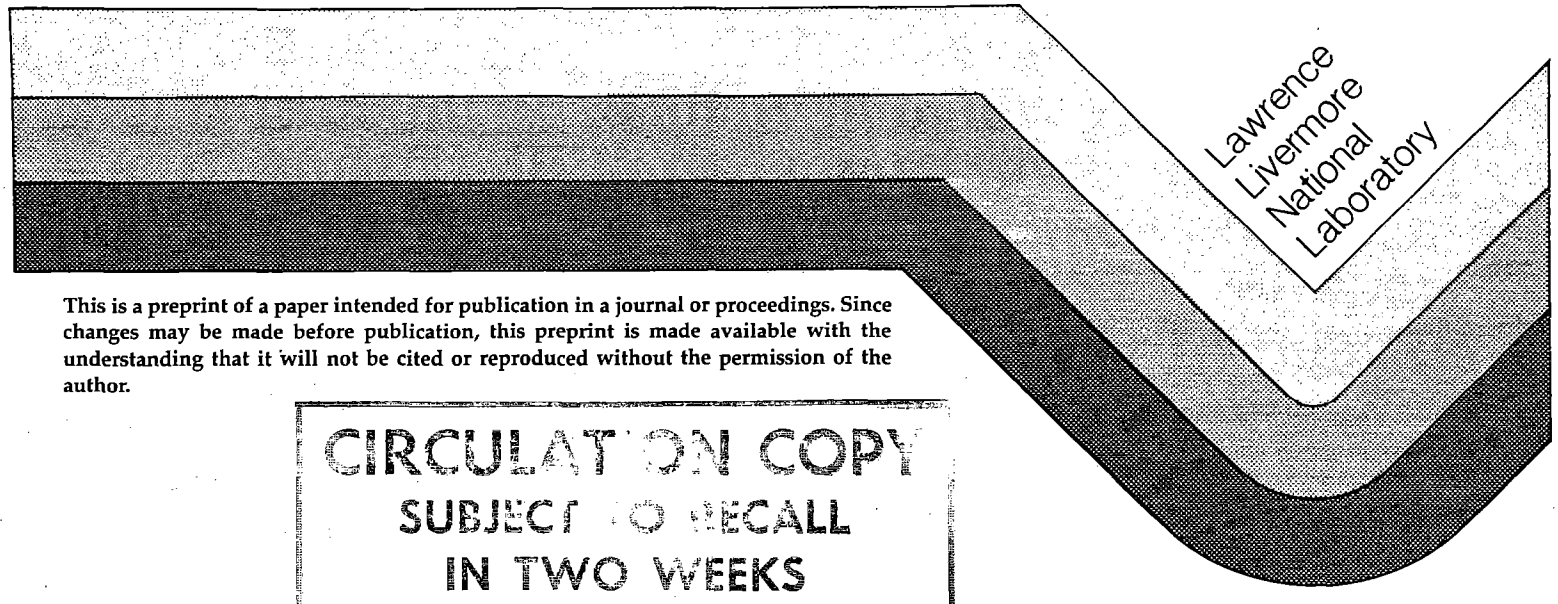
UCRL- JC-107018 Rev. 1
PREPRINT

Retire Fortran? A Debate Rekindled

David Cann
Lawrence Livermore National Laboratory
Livermore, California

This paper was prepared for submittal to
the Supercomputing 1991 Conference,
Albuquerque, New Mexico
November, 1991

July 24, 1991



DISCLAIMER

Retire Fortran? A Debate Rekindled

David Cann
Computing Research Group, L-306
Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, CA 94550
cann@lll-crg.llnl.gov

Abstract

In the May 1984 issue of Physics Today, Jim McGraw debated David Kuck and Michael Wolfe on the question of retiring FORTRAN. They addressed such questions as: Is FORTRAN the best tool for decomposing problems for parallel execution? Is FORTRAN the programming language we should carry into the 21st century? Are there any alternatives? While McGraw argued forcefully in favor of retiring FORTRAN, concerns about performance crippled his position. He could not rebut the claim that only FORTRAN could provide the performance required for scientific computing. In this report, we use the current performance of CRAY SISAL, a functional language for large-scale scientific computing, to counter that claim. If McGraw had had our data in 1984, he could have countered what some say is the only defense in favor of keeping FORTRAN. The results show that we can move beyond the constraints of imperative programming. We can raise the level of abstraction and *retain performance.*

1 Introduction and Motivation

In May of 1984, a debate between Jim McGraw of Lawrence Livermore National Laboratory (LLNL) and David Kuck and Michael Wolfe of Kuck and Associates, appeared in Physics Today [15]. The subject was whether to retire FORTRAN. Seven years have passed, and we wish reopen the debate and provide further evidence that FORTRAN is not the *sine qua non* of high speed computing.

Many believe that the outstanding investments in FORTRAN and the quality of existing FORTRAN compilers is helping to prevent a change in programming methodologies. Many also feel that support for FORTRAN must continue because the language is familiar and widely available. Unfortunately, the complexities of writing correct parallel programs in FORTRAN is a cause of today's software crisis. We believe, as did McGraw in 1984, that increased productivity, generality, utility, portability, and performance are only possible if programmers avoid the constraints of imperative languages and adopt a higher level of abstraction. We must escape the morass of imperative semantics and attain a level of abstraction that separates the programmer from the machine, stresses problem definition over the mechanics of solution, and provides complete information to the compiler.

While Kuck and Wolfe would probably have agreed with the above statements, they wisely argued that programmers simply would not use languages that did not allow them to get the performance they required. To this McGraw had no counterargument for, in 1984, FORTRAN and only FORTRAN provided the performance needed for large-scale scientific computing. The intent of this report is to reopen the debate and present further evidence that functional supercomputing is possible. We compare the execution performance of SISAL, a general purpose functional language for large-scale scientific computing, and FORTRAN on a CRAY X-MP/48. The CRAYs remain the most heavily used machines at our nation's supercomputer centers, and are the machines of choice by most scientific programmers. Thus it is on these machines that languages must beat FORTRAN if they are to replace it.

For completeness and for the benefit of those readers that are not familiar with the principles of functional computing, the next two sections examine the functional programming paradigm, discuss its attributes and advantages, and highlight the salient features of SISAL. In the remaining sections we illustrate the potential inefficiencies of functional computing, present the most recent performance data for the CRAY X-MP, discuss areas of improvement, and make some closing remarks regarding FORTRAN and the future of high speed computing.

2 Programming Alternatives

In 1984, McGraw noted that by all indications future supercomputers would be multiprocessors. Today, most supercomputer users and vendors agree. But can programmers take advantage of the horsepower and not sacrifice portability, longevity and correctness? In this section we highlight the aesthetics of functional programming and contrast the functional model to the imperative model of FORTRAN. To begin, we list the desired characteristics of a true parallel programming language [1]:

1. The language must insulate the programmer from the underlying machine. Deriving and expressing a parallel algorithm is hard enough; one should not have to reprogram it for each new machine.

2. Parallelism must be implicit in the semantics of the language. The compilation system should not have to unravel the behavior of the computation.

3. When a programmer desires determinancy, the language should guarantee it. Regardless of the conditions of execution, a program that realizes a determinate algorithm should yield the same results for the same data.

Of the three items, the last is an issue only when automatic parallelizing compilers are not available and the programmer is responsible for expressing and managing parallelism. Programmers will make mistakes, and these mistakes may remain hidden until system activity changes the rate of execution. This is all we will say about determinancy, as most parallel machines support automatic parallelizing compilers.

Regarding the first two items, however, imperative languages fail to meet the requirements. Remember that languages like FORTRAN were designed to exploit von Neumann machines. As such their computational model assumes that a single program counter will step through a program in the order of the statments within it. This is not necessarily the best path. Because the programmer is responsible for defining the order, he must explicitly map the data to the physical resources of the machine. This mapping is machine dependent, and requires redefinition when the program is moved to a new machine. Further, the choice of mapping affects the compiler's ability to exploit the available parallelism. If the compiler cannot unravel the implied behavior, it cannot remap the data and select the best order of execution. Hence, the compiler is at the mercy of the programmer, and must react conservatively when faced with potential conflicts. The culprit, of course, is the assignment statement.

For example, consider the following FORTRAN excerpt:

```
A = Foo(X)
B = Goo(Y)
```

Determining if these statements can execute in parallel requires a full understanding of both functions. Because of COMMON blocks, they might share data. Further, because of aliasing, some combination of `X`, `Y`, `A`, or `B` might represent the same memory cell. Hence the parallelism in this excerpt is not immediately obvious, and its discovery requires interprocedural analysis or function expansion.

Functional languages, on the other hand, meet all the requirements listed above and do not require analysis for the discovery of parallelism [1,11,13,14]. A functional program is a collection of mathematically sound expressions comprised of both intrinsic and user defined functions. These functions are *well defined* and *determinate*. That is, they define a unique mapping between their domain and their range. A function passed the same set of values will yield the same results regardless of the environment of invocation. This establishes *referential transparency*, which implies that the evaluation of an expression, or the sharing of its subexpressions, does not change the value it denotes. Consequently, expressions are *side effect free*. The concept of a FORTRAN COMMON block does not exist. In the absence of side effects, programmers cannot see the target machine; the concept of *data* replaces memory, and the concept of *creation* replaces update. Further, in the absence of side effects, programs are implicitly parallel.

Now let us return to the code excerpt discussed earlier in this section, but this time in a functional setting. Here `Foo` cannot access the value of `Y` as it is not passed as an argument; and `Goo` cannot access the value of `X` as it is not passed as an argument. Further, `Foo` cannot access `B`, as it is not one of its arguments, or alter `Y`, as it is passed by value. For the same reason, `Goo` cannot access `A` or alter `X`. Since there are no dependencies between `Foo` and `Goo`, they can execute simultaneously. We did not require interprocedural analysis to unravel their behavior. The required information is local and implicit.

3 Sisal Overview

In this section we highlight the salient features of SISAL 1.2 [16] and introduce its soon to be released upgrade (SISAL 2.0). In the interest of promoting the upgrade, we use 2.0 syntax for all the examples in this report. To begin the discussion, we present a SISAL version of Gaussian Elimination without pivoting. It solves a set of linear equations of the form `Ax=B`, where `A` is a `N×M` matrix and `x` and `B` are `N×1` column vectors. Note the Pascal-like syntax and data types.

```
type OneI   = array [..] of integer;
type OneD   = array [..] of double;
type TwoD   = array [..] of OneD;

function Reduce( piv:integer; A:TwoD;
                 B:OneD
                 returns TwoD, OneD )
  let
    % FORM THE MULTIPLIERS FOR THIS
    % REDUCTION STEP
    mults := A[..,piv]/A[piv,piv];
  in
    % REDUCE ALL THE ROWS OF A AT
    % INDEX i (row IS A VECTOR!)
    for row in A at [i] do
      nrow,
      nB := if i = piv then
              % REDUCE THE PIVOT ROW
              row /A[piv,piv],
              B[i]/A[piv,piv]
            else
              % REDUCE A NONPIVOT ROW
              row-mults[i]*A[piv],
              B[i]-mults[i]* B[piv]
            end if
    returns array of nrow,
            array of nb
    end for
  end let
end function

function Gauss( N:integer; A:TwoD; B:OneD
                returns OneD )
  % APPLY N SUCCESSIVE REDUCTIONS TO A
  % AND RETURN THE SOLUTION VECTOR
  for i in [1..N] do
    new A, new B := Reduce( i, A, B );
  returns B
  end for
end function
```

This example shows the two loop forms found in SISAL. The loop in function `Reduce` defines parallel execution while the loop in function `Gauss` defines sequential execution. The absence or presence of reserved word `new` distinguishes between the two. As expected, only the sequential form allows the specification of data dependencies across iterations (`new A` in the example).

Other features of SISAL include reduction operations, records, unions, array modifiers, and streams for non-strict computation. SISAL 2.0 includes several other features, including higher-order functions, true matrices, modules, element wise operations on arrays and streams, and typesets [17]. Of these, higher-order functions are the most important. They add an additional level of abstraction to the model, allowing functions to be treated as data. The next most important are typesets, which allow data types to be derived from argument types and operations that transform types. Both features are important in the development of library software.

4 Problems and Solutions

Earlier in this report we noted that functional languages, because of their semantics, meet all the requirements of a parallel programming language. But by raising the level of abstraction have we hindered performance? We agree with Kuck and Wolfe that performance is the bottom line. Until recently, poor performance has been the strongest argument against functional computing.

Functional languages are considered inefficient because of their side effect free semantics. For example, without side effects an array update must define an entirely new array. However in-place operations can be automatically compiled into most functional programs without violating their computational semantics (called *copy elimination*). The goal is to identify the last consumer of an object and configure it for in-place operation. In this section we illustrate both the costs of copying and the power of copy elimination. To do this we consider the SISAL programs used in this report and present their performance with and without optimization. Table 1 presents the program suite itself. We defer further description of the individual programs until the next section. For now, it will suffice to say that they are all array and floating point intensive, and exhibit parallelism. Table 2 summarizes their performance with and without the copy optimizations. The times are in seconds and were

Table 2: The effectiveness of copy elimination analysis.

Alliant FX/80 Concurrent-Vector (5-ACEs) (IN SECONDS)				
Program	NO Optimization	WITH Optimization	Speedup	% Copies Removed
RICARD	91.0	1.8	50x	99.5%
SIMPLE	1790.9	32.9	54x	99%
WEATHER	1195.3	13.6	87x	96.5%

Table 1: The program suite (C = cycles).

Program	Source Lines	Execution Comments
RICARD	301	200C, 5x1315
SIMPLE	1550	62C, 100x100
WEATHER	2718	10C, 420km

gathered on the Alliant FX/80 at the Northeast Parallel Architectures Center at Syracuse University. We quickly draw two conclusions from the data: The cost of an implementation that naively copies data is truly exorbitant, but the applied optimizations are dramatically effective.

The reader can find a discussion of all the optimizations and their origins in [2,3,5,10,12,18,19]. Further elaboration is beyond the scope of this report. The point we wish to make here is that intelligent compilation can eliminate copying.

5 Potpourri

In the next section we compare FORTRAN to SISAL 1.2 using the CRAY X-MP/48 located at the Open Computer Facility at LLNL. This section enumerates some important considerations, concerns, and caveats, and describes the conditions of the study and the characteristics of the program suite.

5.1 Code Quality

The SISAL compilation system currently generates optimized C as its intermediate code and relies on the local C compiler to finish compilation. This scheme ultimately simplifies code generation and improves portability, but it places the responsibility for code quality on the resident C compiler. Although improving (motivated by the proliferation of UNIX), the quality of most C compilers remains poor. In the study we used OSC version 8.5.A to generate the C, and SCC version 2.0 to compile it. In general, we were impressed with the quality of SCC. Regardless, we do not consider the current SISAL system to be of production quality, although the technology is now ready for industrial transfer. We used CF77 version 4.0.3 to compile the FORTRAN.

5.2 Benchmarking

While there are too many variables to draw definitive conclusions, benchmarking does reveal trends and directions. However, it does not necessarily reflect the true nature of production computing. We feel that this report does provide an accurate comparison of SISAL and FORTRAN.

Each SISAL code in our benchmark suite is a transliteration of its FORTRAN equivalent. At no time were the algorithms changed to improve performance. However, the SISAL codes were written to exploit the true data dependencies of the respective algorithms.

5.3 Levels of Improvement

The data presented in this report does not measure the degree or ease in which tuning could improve both the SISAL and FORTRAN codes. We did not manually annotate the FORTRAN to improve parallel performance. However we did enable as many compiler options as possible to yield the most effective execution. Table 3 shows these options. In general, since most performance tuning involves a restructuring that modifies the flow of data through a computation, we would expect that SISAL would be easier to tune.

5.4 Library Software

Although we only evaluate SISAL's performance in this report, it is important to note that in our work we have addressed the issues of library software and

Table 3: Compilation options for FORTRAN.

CRAY X-MP/48 Unicos 5.1 cf77 4.0.3	
Desired Execution Mode	**Options**
Vector Only	-Zp -Wd"ev68 -dci"
Concurrent Only	-Zp -Wd"ec68i -dv"
Concurrent-Vector	-Zp -Wd"ecv68i"

mixed language programming. These concerns are important in the complete evaluation of a programming language and its ability to perform in an industrial environment. We have implemented a bidirectional interface linking SISAL, FORTRAN, and C, and as a feasibility study, several researchers at LLNL are working to exploit this interface and migrate several production codes into SISAL. For this report, we did not use the interface to access Cray's library software to boost performance.

5.5 Data Gathering

The resident operating system on the OCF CRAY X-MP is Unicos 5.1. The runs were made at off hours and were repeated to insure timing accuracy. We did not accept a parallel run unless we received at least 95% of the machine for the entire execution.

5.6 The Program Suite

We introduced the program suite used in this report in Section 4, but deferred discussion of the individual programs. The smallest code in the suite is RICARD, a production code developed at the University of Colorado Medical Center [6]. RICARD simulates experimentally observed elution patterns of proteins and ligands in a column of gel. For the comparison we ran the first 200 of the 40,000 time steps required to complete a simulation involving 5 proteins and a column of 1,215 levels.

The second largest program is SIMPLE[1], a Lagrangian hydrodynamics benchmark developed at LLNL [8]. This program simulates the behavior of fluid in a sphere. Here we ran a 100 × 100 problem requiring 62 time steps.

The last and largest program is WEATHER [7]. This code, developed at Royal Melbourne Institute of Technology, is a one-level barotropic weather prediction code. The Department of Meteorology at the University of Melbourne provided the original FORTRAN, written and tuned for a VAX uniprocessor; as such, the code is explicitly scalar[2]. Differing from the first two programs, which both occupy a single source file, WEATHER occupies 33 files and manipulates complex data. For this code, we timed the first 10 cycles of execution for a 420 km grid.

The Alliant FX/80 data we provided in the previous section reflects double precision arithmetic (64 bits). The data presented in the next section reflects real arithmetic (64 bits on the CRAY).

6 Performance and Analysis

Tables 4, 5, and 6 present the performance data for all the runs. The first is for vector operation exclusively; the second is for concurrent operation exclusively; and the last is for concurrent and vector operation combined.

The data show two trends. First, the disparity in parallel performance between SISAL and FORTRAN is proportional to the complexity of the program. That is, as the complexity of the application increases so does SISAL's predominance. Observe the four processor concurrent-vector execution times for SIMPLE and WEATHER: Table 6. The SISAL version of WEATHER, which is the most complex program in the suite, yields the best parallel performance improvement over FORTRAN (a speedup of 7.3). The reason for the difference was the lack of interprocedural analysis combined with over-vectorization. For SIMPLE, the second most complex code, the FORTRAN compiler was unable to unravel the true behavior of the equation of state computations found in the inner loops. Thus the SISAL version of SIMPLE ran 2.5 times faster than FORTRAN on four processors.

Second, the quality of the scalar code emitted by SCC is in its current state inferior to CF77. For example, the non-vector execution of RICARD ran 34% faster in FORTRAN than SISAL. This, unfortunately,

[1] Different versions of SIMPLE exist. We used version 2.1 for this report.

[2] Personal communication with Martin Dix, CSIRO Division of Atmospheric Research, Australia.

Table 4: Performance data for the program suite: vector only.

CRAY X-MP/48 Vector (IN SECONDS)		
	One CPU	
Program	Sisal	Fortran
RICARD	0.39	**0.34**
SIMPLE	13.82	**13.03**
WEATHER	**5.94**	13.40

Table 5: Performance data for the program suite: concurrent only.

CRAY X-MP/48 Concurrent (IN SECONDS)				
	One CPU		Four CPUs	
Program	Sisal	Fortran	Sisal	Fortran
RICARD	2.19	**1.44**	0.70	**0.47**
SIMPLE	25.18	**20.57**	17.80	**16.33**
WEATHER	8.74	**7.33**	**2.58**	6.08

Table 6: Performance data for the program suite: concurrent-vector.

CRAY X-MP/48 Concurrent-Vector (IN SECONDS)				
	One CPU		Four CPUs	
Program	Sisal	Fortran	Sisal	Fortran
RICARD	0.39	**0.35**	0.14	**0.11**
SIMPLE	13.52	**13.49**	**4.90**	12.37
WEATHER	**5.93**	13.92	**1.84**	13.51

illustrates one of the few drawbacks of using C as an intermediate form.

In general, the performance data speaks for itself. For the concurrent-vector execution of RICARD, SISAL is competitive. For the concurrent-vector execution of SIMPLE and WEATHER, SISAL is superior. However, there remain two unanswered questions: memory consumption and compilation time. Most skeptics claim that functional languages use memory profligately. In an industrial setting, where every last byte counts, this would prohibit functional computation altogether—regardless of performance. Table 7 shows the mean memory image size of the concurrent-vector runs. Only the SISAL run for WEATHER shows a discrepancy, which we are still investigating. However, the mean memory image for the SISAL version of RICARD is smaller than FORTRAN.

Another complaint against functional languages concerns compilation time. Table 7 shows the compilation time for both languages (for concurrent-vector execution). In general, it appears that the SISAL compiler can generate optimized C code in about the time it takes to compile FORTRAN into executable code. This is reasonable. The use of C as an intermediate form would not be the correct choice for a production compiler. But the data does show that the required optimizations for SISAL are cheap. In fact, their cost is linear in the number of operations in the program (see [9]).

Of course there is room for improvement. All SISAL programs automatically reclaim memory during execution, but the costs can be high. However, the compiler could reduce these associated costs. For example, although we are eliminating most of the intermediate aggregates and unnecessary copying in SISAL programs, we are not currently optimizing the code required to allocate and deallocate arrays between cycles of execution. In most situations, these repetitive operations are unnecessary and the allocation of storage can take place once, before the loop begins execution, and storage can be freed after the loop completes. We refer to this as *aggregate preconstruction*. Profiling the three SISAL programs, we found, respectively, that 9%, 12%, and 18% of the single processor concurrent-vector execution times involved memory management. Note that the disparity between SISAL and FORTRAN for RICARD is just this overhead. We are currently attacking this problem and expect substantial improvements in both sequential and parallel performance of all SISAL programs.

We also expect further improvement when we add unswitching and loop distribution optimizations to the SISAL compiler [20]. The first optimization identifies invariant **if** tests within loops and inverts these computations to hoist the conditionals. For example,

```
for i in [1..n] do
  V := if ( X ~= 0.0 ) then
         A[i]/X
       else
         0.0
       end if;
returns sum V
end for
```

becomes

```
if ( X ~= 0.0 ) then
  for i in [1..n] do
  returns sum A[i]/X
  end for
else
  0.0
end if
```

The second optimization splits loops to isolate non-vector computations. For example, in the following loop the **eos** call prevents vectorization, although it does not inhibit concurrentization:

```
for l in [lm..lx] do
 eosV := eos(eosc,tmp[k,l],rho[k,l]);
 newV := max(tmp[k,l],tflr);
 tmpR := abs((newV-otmp[k,l])/otmp[k,l]);
returns max(tmpR),
        array of eosV,
        array of newV
end for
```

Loop distribution would result in two loops (the first concurrent and the second concurrent-vector):

```
for l in [lm..lx] do
 eosV := eos(eosc,temp[k,l],rho[k,l]);
returns array of eosV
end for

for l in [lm..lx] do
 newV := max(tmp[k,l],tflr);
 tmpR := abs((newV-otmp[k,l])/otmp[k,l]);
returns array of newV, max(tmpR)
end for
```

7 The Evidence Mounts

In this report, we have attempted to rekindle an old debate: Is our reliance on FORTRAN necessary,

Table 7: Miscellaneous data for the program suite: concurrent-vector.

CRAY X-MP/48 Concurrent-Vector (Unicos 5.1)				
	Memory Image Size In Kilowords		Compilation Time In Seconds	
Program	Sisal	Fortran	Sisal to C, C to a.out	Fortran to a.out
RICARD	**93.8**	113.4	6.6, 12.0	12.57
SIMPLE	544.2	**406.0**	42.2, 66.1	35.9
WEATHER	566.0	**163.9**	121.9, 96.3	67.0

or can we transcend the imperative *tar pit* and reach a new level of abstraction? The biggest concern to date has been performance, but recent advances by the functional language community have answered these concerns. The data presented in this report is one such example. Another is the work of Boyle at Argonne National Laboratory who demonstrated parity with FORTRAN using Lisp [4].

We do not claim, however, that SISAL and Lisp are the successors of FORTRAN. What we do claim is that the successor should be functional, and performance is no longer an issue. Tomorrow's parallel machines will not only provide massive parallelism, but will present programmers with a combinatorial explosion of concerns and details. The imperative programming model will continue to hinder the exploitation of parallelism.

In summary, the functional paradigm yields several important benefits. First, programs are more concise and easier to write and maintain. Second, programs are conducive to analysis, being mathematically sound and free of side effects and aliasing. Third, programs exhibit implicit parallelism and favor automatic parallelization. Fourth, programs can run as fast as, if not faster than conventional languages. The time for change is now.

Acknowledgements

Thanks are due to the staff and clients at both the Open Computer Facility at LLNL and the Northeast Parallel Architectures Center at Syracuse University for their patience while we tuned the compiler and collected the data. We also wish to thank Rod Oldehoeft, John Feo, Richard Wolski, and Tom DeBoni for reading and improving this manuscript. Finally, thanks to Jim McGraw, David Kuck, and Michael Wolfe, as their debate in 1984 made this report possible.

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[2] Jeffrey M. Barth. A practical interprocedural data-flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[3] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *The 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, 1989.

[4] J. Boyle and T. Harmer. A practical functional program for the CRAY X-MP. Technical Report MCS-P159-0690, Argonne National Laboratory, 1990.

[5] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[6] J. Cann, E. York, J. Stewart, J. Vera, and R. Maccioni. Small zone gel chromatography of interacting systems: Theoretical and experimental evaluation of elution profiles for kinetically controlled macromolecule-ligand reactions. *Analytical Biochemistry*, (175), December 1988.

[7] P. Chang and G. Egan. An implementation of a barotropic numerical weather prediction model

in the functional language SISAL. In *Proceedings of the SIGPLAN 1990 Symposium on Principles and Practice of Parallel Programming*, March 1990.

[8] W. P. Crowley, C. P. Henderson, and T. E. Rudy. The simple code. Technical Report UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, Februrary 1978.

[9] J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–365, December 1990.

[10] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, 1989.

[11] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[12] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Twelfth Annual ACM Conference of the Principles of Programming Languages*, pages 300–313, January 1985.

[13] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[14] B. J. MacLennan. *Functional Programming Practice and Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[15] J. R. McGraw, D. J. Kuck, and M. Wolfe. A debate: Retire Fortran? *Physics Today*, 37(5):66–75, May 1984.

[16] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[17] R. R. Oldehoeft, D. C. Cann, A. P. W. Böhm, J. T. Feo, and D. H. Grit. SISAL reference manual *language version 2.0.* Technical Report UCRL-JC-104008, Lawrence Livermore National Laboratory, Livermore, CA, December 1988.

[18] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[19] S. K. Skedzielewski and R. J. Simpson. A simple method to remove reference counting in applicative programs. In *Proceedings of CONPAR 88*, September 1988.

[20] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Massachusetts, 1989.